

"Express Mail" mailing label number:

EL675711204US

## SEQUENTIALLY PERFORMED COMPOUND COMPARE-AND-SWAP

Guy L. Steele Jr.,  
Ole Agesen, and  
Nir N. Shavit

### BACKGROUND OF THE INVENTION

#### Field of the Invention

[1001] The invention relates to coordination amongst distributed computational systems and, in particular, to techniques for performing compound operations on memory in linearizable form.

#### Description of the Related Art

[1002] Computer systems often provide primitive instructions or operations that perform compound operations on memory in a linearizable form (i.e., as if atomically). For example, the VAX computer architecture provided instructions to directly support insert and delete operations on double-ended queues. Other processor architectures have provided simpler operations, such as a "test-and-set" operation (e.g., as provided by the IBM 360), a "fetch-and-add" (e.g., as provided by the NYU Ultracomputer), a "load locked" and "store conditional" operation pair (e.g., as provided by the DEC, now Compaq, Alpha), or "compare-and-swap" (e.g., as provided by the Motorola 68000 and processors conforming to the SPARC<sup>TM</sup> architecture):

[1003] SPARC architecture based processors are available from Sun Microsystems, Inc, Mountain View, California. SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems.

[1004] A “compare-and-swap” operation (CAS) typically accepts three values or quantities: a memory address A, a comparison value C, and a new value N. The operation fetches and examines the contents V of memory at address A. If those contents V are equal to C, then N is stored into the memory location at address A, replacing V. Whether or not V matches C, V is returned or saved in a register for later inspection (possibly replacing either C or N, depending on the implementation). All this is implemented in a linearizable, if not atomic, fashion. Such an operation may be notated as “CAS(A, C, N)”.

[1005] A more powerful and convenient operation is “double compare-and-swap” (DCAS), which accepts six values: memory addresses A<sub>1</sub> and A<sub>2</sub>, comparison values C<sub>1</sub> and C<sub>2</sub>, and new values N<sub>1</sub> and N<sub>2</sub>. The operation fetches and examines the contents V<sub>1</sub> of memory at address A<sub>1</sub> and the contents V<sub>2</sub> of memory at address A<sub>2</sub>. If V<sub>1</sub> equals C<sub>1</sub> and V<sub>2</sub> equals C<sub>2</sub>, then N<sub>1</sub> is stored into the memory location at address A<sub>1</sub>, replacing V<sub>1</sub>, and N<sub>2</sub> is stored into the memory location at address A<sub>2</sub>, replacing V<sub>2</sub>. Whether or not V<sub>1</sub> matches C<sub>1</sub> and whether or not V<sub>2</sub> matches C<sub>2</sub>, V<sub>1</sub> and V<sub>2</sub> are returned or saved in registers for later inspection. All this is implemented in a linearizable, if not atomic, fashion. Such an operation may be notated as “DCAS(A<sub>1</sub>, A<sub>2</sub>, C<sub>1</sub>, C<sub>2</sub>, N<sub>1</sub>, N<sub>2</sub>)”.

[1006] The SPARC version 9 architecture supports an implementation of CAS instruction on both 32-bit-wide and 64-bit-wide operands, but does not provide a DCAS instruction. On the other hand, the Motorola 68040 processor supports a double compare-and-swap instruction (on the 68040 it is called “CAS2”). Unfortunately, the CAS2 instruction effectively locks the entire memory system rather than locking first one location and then another. The net result is that deadlock is not possible but CAS2 instructions executed by separate processors are always serialized in time even if they access distinct memory locations.

[1007] There is substantial theoretical literature on the use of CAS operations and ordinary DCAS operations in the design of non-blocking algorithms. For example, Massalin and Pu proposed a collection of DCAS-based concurrent algorithms. They built a lock-free operating system kernel based on the DCAS operation offered by the Motorola 68040 processor, implementing structures such as stacks, FIFO-queues, and

[1009] A drawback of the DCAS operation (as implemented by the Motorola 68040 processor and as described in the literature) is that it requires both of memory addresses,  $A_1$  and  $A_2$ , to be valid. If either of the memory addresses  $A_1$  or  $A_2$  is invalid, then a memory fault occurs, typically resulting in an interrupt or trap. For certain kinds of algorithms that could otherwise profitably make use of the DCAS operation, such behavior is undesirable. For example, under certain conditions, the memory address  $A_2$  might be invalid or unusable, but only when the contents of memory address  $A_1$  are unequal to  $C_1$ .

[1011] In general, two or more storage locations are accessed by compound synchronization operations in accordance with the present invention. Ordered locking

of storage locations, if provided, is in accordance with any fixed total order of storage location. For example, ascending memory address order or descending memory address order are both suitable fixed total orders. While realizations in accordance with the present invention typically provide coordination between concurrently executed instances of a particular compound synchronization instruction (e.g., a DCAS), coordination between differing instructions (e.g., between 2-way and 3-way sequentially performed nCAS instructions) or between differing instruction types (e.g., between an nCAS instruction and a corresponding sequentially performed compound synchronization operation of some other type) may be desirable in some implementations. Indeed, the use of a compare-and-swap synchronization primitive is merely illustrative. Other realizations in accordance with the present invention may build on other synchronization or access primitives, e.g., read-modify-write, read-compute-conditionally write, test-and-set, etc. Similarly, while the exemplary sequentially performed DCAS and nCAS operations described herein employ the same type of access primitive in each leg thereof (i.e., a compare-and-swap), other realizations need not. For example, sequentially performed compound synchronization operations are envisioned in which earlier- and later-reserved storage locations may be accessed using different access primitives.

[1012] In some implementations, locking of memory locations is provided on a cache line basis. In others, other reservation schemes (e.g., per location locking) may be provided. More generally, reservation may include any implementation-appropriate mechanism for obtaining some form of exclusive access or rights with respect to the reserved object. In some realizations of a compound synchronization operation in accordance with the present invention, the signaling of a fault corresponding to a later reserved memory location depends on a value read from an earlier reserved memory location. For example, in some realizations of an nCAS operation, signaling of a fault corresponding to a later reserved memory location depends on successful comparison of a value read from an earlier reserved memory location with a corresponding test value.

[1013] Processors, instruction sets, computer systems and computer program products implementing and/or exploiting such sequentially performed compound compare-and-

### **BRIEF DESCRIPTION OF THE DRAWINGS**

[1018] The use of the same reference symbols in different drawings indicates similar or identical items.

### **DESCRIPTION OF THE PREFERRED EMBODIMENT(S)**

- 5 -

[1020] In general, a coherent presentation of memory space 102 and its component storage locations is maintained using any suitable techniques and/or coherence protocols. In the illustrated configuration, interactions between processors 101 and memory space 102 are mediated by respective caches 103. Accordingly, in an exemplary realization in accordance with FIG. 1, “cache-coherent” means that, from the perspective of each processor and its associated memory cache, the shared memory is logically divided into portions of, for example, 128 bytes (called “cache lines”) and that the system includes a memory controller protocol that allows each processor’s cache either to “own” a cache line or to “share” a given cache line. Only one processor cache at a time may own a cache line, but any number of processors may share a cache line that is not owned. When a processor cache acquires ownership of a cache line, a copy of the data in that cache line is delivered to the cache of that processor. A processor whose cache owns a cache line is permitted to read and write the data for that cache line using the copy in its cache. If a processor whose cache owns a cache line changes the copy of the data in its cache, then the updated data must be transferred back to the shared memory at some point before ownership is released. A processor whose cache shares a cache line is permitted to read its copy of the cache line data but not to update it.

[1021] During execution of most instructions, the caches of respective processors acquire and release sharing or ownership of cache lines automatically as necessary to accomplish the reading and writing of data that may be requested by the processors. However, it is also possible for a processor to request its cache to acquire ownership of a cache line and to refuse to yield ownership until its processor directs otherwise. Such a request from a processor to its cache is called “locking” a cache line, and permission from the processor to its cache to yield ownership of a locked cache line is called “unlocking” the cache line.

- 6 -

### Sequentially Performed DCAS

[1024] In response to a sequential performed DCAS instruction, DCAS( $A_1, A_2, C_1, C_2, N_1, N_2$ ), processor 101 behaves as follows:

- 7 -

**FIG. 3.** The branch to step 3 and related flows are detailed in **FIG. 3**, while the branch to step 5 and related flows are summarized (302).

2. Attempt to lock the cache line that includes the memory words identified by addresses  $A_1$  and  $A_2$ . If the cache line is being shared or is owned by another processor, the request will stall until the cache line becomes available. If it is necessary to interrupt this processor during a stall, then the program counter is left pointing to the sequential DCAS instruction, as if execution of this instruction had not commenced. If the memory address  $A_1$  is invalid, then no cache line ownership is obtained and a memory fault is signaled in the customary manner for address  $A_1$ , resulting in a trap or interrupt that terminates execution of the instruction. If the memory address  $A_2$  is invalid, then set A2OKAY to false; otherwise set A2OKAY to true. Go to step 7.
3. Attempt to lock (303) the cache line that includes the memory word identified by address  $A_1$ . If the cache line is being shared or is owned by another processor, the request will stall (304) until the cache line becomes available. If it is necessary to interrupt (305) processor 101 during a stall, then the program counter is left pointing to the sequential DCAS instruction, as if execution of the instruction had not commenced. If the memory address  $A_1$  is invalid, then no cache line ownership is obtained and a memory fault is signaled (307) in the customary manner for address  $A_1$ , resulting in a trap or interrupt that terminates execution of the instruction. Continue to step 4.
4. Attempt to lock (308) the cache line that includes the memory word identified by address  $A_2$ . As before, if the cache line is being shared or is owned by another processor, the request will stall (309) until the cache line becomes available. If it is necessary to interrupt (310) this processor during a stall, then the cache line that was locked at 303 is unlocked (311) and the program counter is left pointing to the sequential DCAS instruction, as if execution of this instruction had not commenced. If the memory address  $A_2$  is invalid, no reservation is performed and no memory fault is signaled, but A2OKAY is set (312) to false. Otherwise A2OKAY is set to true (313). Go to step 7.
5. Attempt to lock the cache line that includes the memory word identified by address  $A_2$ . If the cache line is being shared or is owned by another processor,



the request will stall until the cache line becomes available. If it is necessary to interrupt this processor during a stall, then the program counter is left pointing to the sequential DCAS instruction, as if execution of this instruction had not commenced. If the memory address  $A_2$  is invalid, no reservation is performed and no memory fault is signaled, but A2OKAY is set (314) to false. Otherwise A2OKAY is set to true. Continue with step 6.

6. Attempt to lock the cache line that includes the memory word identified by address  $A_1$ . If the cache line is being shared or is owned by another processor, the request will stall until the cache line becomes available. If it is necessary to interrupt this processor during a stall, then the cache line that was locked in step 5 is unlocked and the program counter is left pointing to the sequential DCAS instruction, as if execution of the instruction had not commenced. If memory address  $A_1$  is invalid, then no cache line ownership is obtained and a memory fault is signaled (315) in the customary manner for address  $A_1$ , resulting in a trap or interrupt that terminates execution of the instruction. Continue with step 7.
7. Fetch (316) from the cache of processor 101 the contents  $V_1$  of the word at memory address  $A_1$ . Continue with step 8.
8. Compare (317)  $V_1$  to  $C_1$ . If they differ, then replace  $N_1$  with  $V_1$  and replace  $N_2$  with zero (318), and go to step 13. Otherwise continue with step 9.
9. If A2OKAY is false, signal (319) a memory fault for address  $A_2$  in the customary manner, resulting in a trap or interrupt that terminates execution of the instruction. Otherwise, continue with step 10.
10. Fetch (320) from the cache of processor 101 the contents  $V_2$  of the word at memory address  $A_2$ . Continue with step 11.
11. Compare (321)  $V_2$  to  $C_2$ . If they differ, then replace  $N_1$  with  $V_1$  and replace  $N_2$  with  $V_2$  (322), and go to step 13. Otherwise continue with step 12.
12. Store (323) to the cache of processor 101 the value  $N_1$  into the memory word at address  $A_1$  and likewise store (323)  $N_2$  into the memory word at address  $A_2$ . Replace  $N_1$  with  $V_1$  and replace  $N_2$  with  $V_2$ . Continue with step 13.

13. Unlock (325) the cache line that includes the memory word identified by address  $A_1$ . If SHARED is false and A2OKAY is true, unlock (326) the cache line that includes the memory word identified by address  $A_2$ . Terminate execution of the instruction.

[1025] Based on the description herein, persons of ordinary skill in the art will appreciate that steps 1 through 6, above, reserve locations associated with addresses  $A_1$  and  $A_2$  in an order such that if the memory locations identified by these addresses belong to two different cache lines then the lower address is reserved first. This technique avoids a deadlock situation in which two separate processors each attempt to reserve two memory locations P and Q, but one processor reserves P first and then Q while the other reserves Q first and then P, allowing the possibility of each processor succeeding at its first reservation request and then stalling, waiting for the other processor to release its reservation.

[1026] While the above description has presumed use of cache line locking, other reservation mechanisms may also be employed. For example, locking may be performed on an individual storage location basis. Similarly, although memory address order (e.g., ascending or descending) is simple and multiple memory locations can be efficiently ordered in accordance therewith, other ordering techniques are also possible. In general, any fixed total order of storage locations employed by respective instances of sequentially performed compound synchronization operations is suitable.

[1027] In some realizations, deadlock avoidance need not be provided by the implementation of the sequentially performed DCAS instruction. For example, in one such realization, steps 5 and 6 (above) may be omitted and step 1 is updated to change the words "step 5" to "step 3." Accordingly, the DCAS implementation itself no longer discriminates based on the ordering of  $A_1$  and  $A_2$  with respect to some fixed total order of memory locations. Deadlock is a possibility and it is, instead, up to the programmer to use the DCAS instruction carefully to avoid deadlock. For example, code employing such a modified DCAS instruction may employ preceding instructions to ensure that operands are supplied to the DCAS instruction in accordance with a fixed total order of memory locations. In any case, other

[1028] In some realizations, a sequentially performed DCAS instruction may also deliver a Boolean (single-bit) result to indicate succinctly whether or not data was successfully stored to memory. For example, steps 8 and 11 (above) may be suitably augmented by inserting “store FALSE into the Boolean result register” before the words “go to step 13.” Similarly, step 12 (above) may be suitably augmented by inserting “Store TRUE into the Boolean result register” before the words “Continue with step 13.”

[1029] While a double compare-and-swap instruction serves as a useful example, persons of ordinary skill in the art will appreciate that the techniques described herein may be extended to other instructions including instructions that act upon a larger number of memory locations. For example, a sequential compound compare-and-swap (nCAs) accepts  $n$  memory addresses  $A[j]$  ( $1 \leq j \leq n$ ),  $n$  comparison values  $C[j]$  ( $1 \leq j \leq n$ ), and  $n$  new values  $N[j]$  ( $1 \leq j \leq n$ ).

1. Grade the memory addresses in numerical order (as if by the “grade up” operation of Iverson's APL programming language) so as to produce a grade vector  $G$  that is a permutation of the integers from 1 to  $n$  such that, for all  $1 \leq j < n$ ,  $A[G[j]] \leq A[G[j+1]]$ .

If  $j = 1$ , set  $SHARED[j]$  to false. If  $j > 1$  and memory addresses  $A[G[j-1]]$  and  $A[G[j]]$  both identify memory locations that belong to the same cache line, then set  $SHARED[j]$  to true and copy  $AOKAY[G[j-1]]$  into  $AOKAY[G[j]]$ . Otherwise, set  $SHARED[j]$  to false and ask this processor's cache to lock the cache line that includes the memory word identified by address  $A[G[j]]$ . If the cache line is being shared or is owned by another processor, the request will

stall until the cache line becomes available. If it is necessary to interrupt this processor during a stall, then (first) for all values of  $k$  in order from 1 to  $j-1$ , if  $SHARED$  is  $FALSE$  and  $AOKAY[G[j]]$  is  $TRUE$ , the cache line that includes the memory word identified by address  $A[G[k]]$  is unlocked; and (second) the program counter is left pointing to the sequential  $nCAS$  instruction, as if execution of this instruction had not commenced. If the memory address  $A[G[j]]$  is invalid, no reservation is performed and no memory fault is signaled, but  $AOKAY[G[j]]$  is set to false. Otherwise  $AOKAY[G[j]]$  is set to true.

3. Repeat the following for all values of  $j$  in order from 1 to  $n$ :

If  $AOKAY[j]$  is false, signal a memory fault for address  $A[j]$  in the customary manner, resulting in a trap or interrupt that terminates execution of the instruction. Otherwise, fetch (from this processor's cache) the contents  $V[j]$  of the word at memory address  $A[j]$ . Then, compare  $V[j]$  to  $C[j]$ . If they differ, then set  $L$  to  $j-1$ , terminate the repetition of this step, and go to step 6.

4. Repeat the following for all values of  $j$  in order from 1 to  $n$ :

Store (to this processor's cache) the value  $N[j]$  into the memory word at address  $A[j]$ .

5. Set  $L$  to  $n$ .

6. Repeat the following for all values of  $j$  in order from 1 to  $L$  (if  $L$  equals zero, do not perform this step at all):

Replace  $N[j]$  with  $V[j]$ .

7. Repeat the following for all values of  $j$  in order from  $L+1$  to  $n$  (if  $L$  equals  $n$ , do not perform this step at all):

Replace  $N[j]$  with zero.

8. Repeat the following for all values of  $j$  in order from 1 to  $n$ :

If  $SHARED[j]$  is false and  $AOKAY[G[j]]$  is true, unlock the cache line that includes the memory word identified by address  $A[G[j]]$ .

[1031] Note that nCAS never locks any given cache line more than once and that it locks them in ascending address order. In one variation, each occurrence of "SHARED[j]" (in steps 2 and 8, above) may be replaced by "SHARED[G[j]]".

[1032] As before, the sequential nCAS instruction may be augmented to deliver a Boolean (single-bit) result to indicate succinctly whether or not data was successfully stored to memory. For example, step 3 (above) may be augmented by inserting "store FALSE into the Boolean result register" before the words "go to step 6", and step 5 may be augmented by inserting "Store TRUE into the Boolean result register." before the words "Set L to n."

[1033] Also as before, operation of a modified sequential nCAS instruction may omit the ordering of memory addresses before locking the cache lines and instead lock in the order in which the addresses are presented. For such a modified instruction, it is as if the vector G were always treated as consisting of the integers from 1 through n in order, rather than as a vector that indicates how to sort the vector A of addresses. As before, the burden of deadlock avoidance may be shifted to the programmer.

[1034] While the above description of sequentially performed DCAS and nCAS instructions has presumed that signaling of faults corresponding to a later reserved memory location is entirely dependent on success of an access to an earlier reserved memory location, variants of the sequentially performed DCAS or nCAS instruction may distinguish more than one category of invalid or unusable memory address and signal (or not) accordingly. For example, a memory fault of one category may be selectively signaled in accordance with the outcome of comparison operations as outlined above, whereas a memory fault of another category is always signaled regardless of the outcome of the comparison operations. In one exploitation, presentation of a memory address that is a multiple of 4 (i.e., properly aligned), but for which there is no entry in a virtual page table might trigger a memory fault only if the memory contents of all preceding addresses equal their corresponding memory values, whereas presentation of a memory address that is not a multiple of 4 could always signal a memory fault.

[1035] While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that

the scope of the invention is not limited to them. Many variations, modifications, additions, and improvements are possible. For example, compare-and-swap synchronization is only one suitable primitive form. Other compound synchronization operations may employ (i) fault signaling for an access to a later reserved storage location, which depends on an access to an earlier reserved storage location, and/or (ii) reservation of such storage locations in accordance with a fixed total order thereof. Such compound synchronization operations may employ other access primitives such as test-and-set, read-modify-write, read-compute-conditionally write, etc. In addition, reservation of storage locations may be performed by locking associated cache lines or otherwise, such as by locking other groupings or by locking storage locations individually. Some realizations need not employ cache memory as an intermediary or locking facility. In general, any fixed total order of storage locations, including ascending and descending memory address orders, may be employed. Although certain exemplary compound synchronization instructions have been described that operate on shared byte-addressable memory, variations that employ other shared storage locations such as shared registers are also envisioned.

[1036] More generally, plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of claims that follow. Structures and functionality presented as discrete components in the exemplary configurations may be implemented as a combined structure or component in other configurations. These and other variations, modifications, additions, and improvements may fall within the scope of the invention as defined in the claims that follow.